УДК 004.415.53

А.Л. Єрохін[1], М.О. Мончак[2]

[1]Харківський національний університет внутрішніх справ, Харків
[2]Харківський національний університет радіоелектроніки, Харків

МЕТОД ПЕРЕТВОРЕННЯ АРХІТЕКТУРИ БІЗНЕС-ЛОГІКИ В СЛАБОСТРУКТУРОВАНИХ ПРОГРАМНИХ СИСТЕМАХ

У статті запропоновано метод перетворення архітектури бізнес-логіки в слабоструктурованих системах до бажаного архітектурного стану. Метод включає методику виявлення необхідності перетворень, критерії, які дозволяють спроектувати більш вдалу архітектуру та опис архітектурного рефакторингу, за допомогою якого можливо здійснити необхідні перетворення.

Ключові слова: *архітектура системи, структура програми, архітектурний рефакторинг*

Вступ

В останні роки спостерігається тенденція до збільшення тривалості життєвого циклу успішних програмних проектів. Як наслідок, зростає обсяг успадкованого коду, що підтримується спільнотою розробників [1]. Саме це пояснює виняткову важливість завдань, пов'язаних з полегшенням супроводу та розвитку існуючого програмного коду. У той же час, цим задачам приділяється недостатня увага з боку наукового співтовариства і розробників інструментальних засобів. Як наслідок, сучасні методики переоцінюють значення початкової фази життєвого циклу програмної системи і практично ігнорують її подальшу еволюцію. Таким чином, в даний час існує явна потреба в методиках і ефективних інструментах підтримки роботи з існуючим кодом.

Останнім часом намітився перелом ситуації: стали викликати значний інтерес питання систематичного використання трансформацій як центрального принципу процесу розвитку та супроводження існуючого програмного забезпечення (ПЗ). Однак більшість дослідників розглядає трансформації досить вузько – як трансформації на рівні вихідного коду – рефакторинг [2]. Тим не менше, в даний час практично не існує досліджень, присвячених трансформації на більш високому рівні абстракції – рівні архітектури ПЗ. У той же час, існує багато сценаріїв супроводу та розвитку існуючого коду де метою є зміна архітектури існуючої системи. У зв'язку з цим, великий інтерес викликає розробка методики та супроводжуючих її інструментальних засобів, спрямованих на організацію передбачуваною і керованого процесу зміни архітектури ПЗ.

Для вирішення проблеми незрозумілого і слабоструктурованого коду можна запропонувати два методи. Перший – дорогий, складний і ризикований. Це позбавлення від всього незрозумілого слабкоструктурованого коду та написання замість нього зрозумілого. Переваги такого підходу: можна зробити все так, що воно буде найкраще відповідати усім бажаним функціям системи, оскільки робиться все «з нуля». Недоліки: для виконання такої процедури необхідно повністю усвідомити, що і як робить код, далі необхідно повністю припинити його розвиток і на невизначений термін зайнятися виконанням наново тієї роботи, яка вже колись була зроблена. Це дуже дорогий в матеріальному плані метод, бо розробники витратять масу часу для його здійснення. До того ж, при виконанні таких операцій, не можна буде зупинитися, не закінчивши їх – в такому випадку отриманий код буде непридатний для використання. Також, з огляду на відсутність автоматичного тестування, таке завдання може виявитися нездійсненним.

Існує й інший варіант вирішення цієї проблеми – використання методів рефакторинга. Цей варіант не передбачає ні ідеального знання перетворює коду, ні припинення основної роботи над проектом. Рефакторинг займає набагато менше часу, не вимагає свого повного завершення. Відсутність автоматичного тестування, звичайно, знову ж таки ускладнює завдання, але рефакторинг цілком припускає введення такого при виконанні самого рефакторингу.

В даний час є досить цікаві напрацювання з теорії рефакторингів, однак дуже мало досліджень щодо застосування їх у промисловому програмуванні на реальних проектах.

Виходячи з вищезазначеного, сформулюємо задачі дослідження. В даній роботі необхідно:

– дослідити прояви слабоструктурованого коду;

– запропонувати підхід до розробки структурованої архітектури;

– розробити метод перетворення слабоструктурованого коду до бажаного архітектурного стану.

Дослідження симптомів слабоструктурованого коду

Потреба у зміні існуючого коду програмного забезпечення (надалі ПЗ) може виникнути в ході вирішення широкого кола завдань по його модернізації. У загальному випадку зміни здатні торкнутися не тільки коду, але й всіх інших складових, пов'язаних з програмною системою що трансформується. Одним із найбільш істотних різновидів змін ПЗ є зміна архітектури програмної системи. В якості

 © А.Л. Єрохін, М.О. Мончак

прикладів можна навести такі сценарії, які потребують зміни архітектури існуючого ПЗ:

а) перетворення, зумовлені функціональними змінами ПЗ. Бажано, щоб впровадження нової функціональності не торкнулося існуючої логіки системи. Також бажано, щоб складність впровадження нової функціональності в існуючу систему не перевищувало істотним чином складність реалізації цієї функціональності в рамках нового проекту. Гарна архітектура дозволяє досягти поставлених цілей. Отже, зміна існуючої архітектури – хороший крок на шляху впровадження нової функціональності, до того ж такі зміни полегшують і подальшу еволюцію системи;

б) зміна платформи ПЗ. Вкрай бажано, щоб зміна платформи ПЗ якомога менше торкнулася існуючого коду, і щоб можна було обмежитися змінами тільки у вузькому платформо-залежному прошарку системи. Виділення такого прошарку – архітектурне завдання. Його рішення завжди пов'язане з необхідністю зміни архітектури;

в) перетворення, пов'язані з реорганізацією компанії, яка веде розробку. Прикладом, такої реорганізації може стати зовнішня розробка (аутсорсінг). Рішення про використання аутсорсінгу – типовий крок по оптимізації виробництва. Нажаль, цей крок часто ускладнюється проблемою виділення і передачі компонентів для зовнішньої розробки. Зміна архітектури програмної системи здатна полегшити вирішення цієї задачі.

Список сценаріїв, які призводять до потреби в зміні архітектури існуючого ПЗ, на цьому не вичерпується: наведені вище приклади покликані лише продемонструвати широкий спектр завдань, які обумовлюють необхідність подібних змін.

Незважаючи на те, що всі програми різні, проблеми, які в них виникають, досить спільні. Розглянемо перераховані проблеми коду web-додатків, які можна найчастіше зустріти:

а) відсутність дизайну високого рівня. В такому випадку немає ніяких рекомендацій щодо того, як потрібно писати код, яких принципів і підходів дотримуватися. Кожен розробник обирає свою власну стратегію. Це, в свою чергу, призводить до таких проблем: труднощі розуміння проекту в цілому, труднощі розуміння реалізації окремих частин програми, складності прогнозування нових розробок. Нові розробки, поліпшення проекту, додавання нових функціональностей дуже важко піддаються оцінці, дуже складно заздалегідь визначити скільки часу буде займати їхня реалізація і як це буде здійснено;

б) відсутність прошарку бізнес-логіки. Як кажуть програмісти, бізнес-логіка «розпливається» по проекту, вона реалізована в різних частинах програми (на рівні відображення, у допоміжних класах, в збережених процедурах бази даних тощо). Це призводить до таких проблем: додаткове навантаження на базу даних, що призводить до втрати продуктивності; повторення функціональності в збережених процедурах; складність пошуку де і як реалізована бізнес-логіка; деякі важливі бізнес-правила можуть бути загублені в окремих функціональностях; дані, які не пройшли необхідну перевірку, можуть потрапити в базу даних; дивлячись на код складно зрозуміти як реалізоване конкретне бізнес-завдання;

в) низький рівень повторного використання коду. Більша частина функціональності реалізована в елементах управління користувача або прямо на сторінках, що значно обмежує область її використання. Це призводить до так званого «copy-paste» програмування, що потенційно може викликати великі проблеми;

г) повторення коду та функціональності. Часто дуже схожі ділянки коду зустрічаються в різних частинах програми. Це ускладнює пошук ділянок програми, де повинні бути зроблені зміни, створюється небезпека того, що зміни будуть зроблені не у всіх місцях, де це необхідно;

д) велика кількість допоміжних класів, які ніяк не організовані і не пов'язані один з одним. Ці класи створюються лише для того, щоб забезпечити доступ до окремих методів з різних частин програми. З такими класами важко працювати, тому що вони ніяк не систематизовані. Це викликає повторення коду, модифікацію коду тільки в одному класі. Такі класи спричиняють доступ до коду в тих місцях, де цього не повинно бути;

е) великі класи, так звані „класи – Боги”. Це класи, які мають декілька зон відповідальності. Часто вони містять код декількох окремих розробок. Такі класи важко розуміти і працювати з ними, часто вони виконують непотрібні дії і звернення до бази даних, такі класи неможливо покрити автоматичним тестуванням;

ж) проблема «стрільби дробом». Часто, для реалізації невеликих змін, необхідно внести багато маленьких змін в різні частини програми (збережені процедури, сторінки, елементи управління, допоміжні класи і т.д.). Це призводить до труднощів пошуку таких частин. При внесенні подібних змін багато сторінок повинні бути заново протестовані;

з) довгі методи. Деякі частини програми реалізовані як довгі методи високої складності, з безліччю циклів, переходів, галужень. Це ускладнює розуміння алгоритму методу та можливих шляхів виконання, ускладнює написання автоматичних тестів для такого методу, через підвищену складність методу велика ймовірність виконання марних дій, що приводить до погіршення продуктивності;

і) висока взаємозалежність класів. Іноді класи настільки залежать один від одного, що використання одного з них, без використання іншого виявляється неможливим. Такі класи навіть іноді знаходяться в одному файлі, для виконання необхідної операції приходиться виконувати ряд марних, функціональність одного класу не може бути протестована без залучення іншого, класи, як правило, мають складний для розуміння взаємозв'язок;

к) змішування даних. Дані, що зберігаються в базі даних у нормалізованому вигляді, часто змішуються і представляються елементам управління у вигляді суміші всіх необхідних даних. Це призводить до неефективного кешування даних і до попадання в кеш повторюваних даних. Одні й ті ж самі дані можуть бути використані в програмі в різних форматах і з різних джерел;

л) доступність даних всім частинам програми. Всі дані можуть бути модифіковані будь-якою частиною програми. Це ускладнює пошук місця програми, де дані були невірно модифіковані, робить неможливим визначення від яких даних залежить клас;

м) приховані операції. Деякі класи виконують приховані дії. Вони можуть створювати екземпляри нових класів, робити неявні запити до бази даних та інші небезпечні дії. Розробники часто не підозрюють про такі операції, що призводить до втрати продуктивності.

На цьому перелік проблем коду не закінчується, є ще багато інших показників якості програмного коду [2].

Але слід зазначити, що неможливо чітко визначити, коли потрібно починати рефакторинг, а коли слід зупинитися. Це досить суб'єктивний процес, який базується в основному на людській інтуїції, спроектованій на інформацію.

Підходи до розробки структурованої архітектури

Перед тим, як почати розробляти нову, більш підходящу архітектуру, слід чітко визначити цілі нової архітектури. Найбільш поширені цілі зміни архітектури є такими:

а) створення високорівневої структури програми, що відповідає загальноприйнятим рекомендаціям. Створення програми, чітко розділеної на шари. Програма має містити три рівні абстракції: рівень відображення, рівень бізнес-логіки та рівень доступу до даних. Така архітектура є проявом відомого шаблону проектування, під назвою MVC (Model-View-Controller) [3];

б) поліпшення розширюваності програми. Архітектура повинна бути такою, щоб додавання нових функцій було можливе без порушення початкової структури, з найменшим впливом на існуючу функціональність, за найкоротші терміни і максимально незалежно від будь-яких інших змін;

в) спрощення повторного використання вже існуючої функціональності. Хороша структура повинна дозволяти використовувати повторно будь-яку ділянку коду користуючись виключно методиками ООП (такими як рефакторинг). Неприпустимо копіювання коду в інші частини програми. На великому проекті, існує ймовірність того, що реалізація нової функціональності цілком може полягати в правильній комбінації існуючих і це не призведе до порушень в роботі існуючого коду;

г) підвищення модульності програми. По-справжньому гнучка програма повинна бути представлена набором модулів. Кожен модуль повинен бути максимально незалежним. Це означає, що кожен модуль повинен представляти сутність предметної області, повинен мати простий інтерфейс і ясний набір завдань. Залежності між модулями повинні бути мінімізовані. У цьому випадку бізнес-логіка програми повинна бути відділена від представлення, що дозволить досить просто змінювати інтерфейс користувача. Такий підхід також спрощує взаємодію зі сторонніми програмними модулями, які використовують подібний підхід;

д) підвищення стабільності коду. Строга типізація повинна застосовуватися всюди, де це можливо для виявлення помилок на етапі компіляції, а не виконання. Будь-яка функціональність не повинна бути доступна з ділянок коду, де вона може бути використана некоректно;

е) поліпшення продуктивності. Найчастіше, основним «вузьким місцем» програми є база даних. Саме з нею пов'язані основні проблеми з продуктивністю. Тому, необхідно якомога більше полегшити навантаження на базу даних. База даних повинна перетворитися виключно в сховище даних. Будь-яка логіка, яка знаходиться на рівні бази даних, повинна бути винесена на рівень бізнес-логіки програми, всі запити до бази даних повинні кешуватися. Бізнес-логіка повинна бути спроектована таким чином, щоб ніколи не виконувався той код, в якому немає необхідності. Результати обчислень повинні зберігатися замість того, щоб обчислюватися знову;

ж) полегшення підтримки коду. Код повинен бути таким, що самодокументується. Імена класів та методів повинні чітко відображати те, що вони роблять, щоб у розробника не виникало необхідності дивитися на їх реалізацію. Коментарі та діаграми повинні бути використані для опису особливо складних ділянок програми. Це дозволить забезпечити більш ефективний обмін знаннями серед розробників, що в свою чергу допоможе позбавитися від незамінних членів команди;

з) підвищення безпеки. Доступ до конфіденційної інформації повинен суворо контролюватися в коді.

Розробка структури системи є досить типовою задачею, жодна програма не обходить цей етап. Тому існують основні принципи, керуючись якими можна створити структуровану архітектуру програми. Основні принципи:

а) поділ на прошарки. Будь-яке нове архітектурне рішення має бути узгоджене з високорівневою структурою прошарків. Одна і та ж архітектурна модель повинна використовуватися у всіх компонентах системи;

б) декомпозиція. Будь-яка складна функціональність повинна бути розбита на більш дрібні. Те ж стосується і сутностей. Наприклад, деякий об'єкт повинен містити дані деякої сутності, потім ще один об'єкт або їх набір повинен містити операції, здійс-

ненні над даною сутністю, і, ще один об'єкт повинен містити алгоритм обробки;

в) інкапсуляція. Дуже складно зрозуміти всю функціональність величезної програми. На щастя, необов'язково знати всю функціональність окремих класів, їх реалізацію, якщо відомо їхні функції. Програміст повинен вміти користуватися будь-якими класами, не вдаючись у деталі їхньої реалізації, а класи повинні дозволяти це;

г) уникнення сильних зв'язків. Кожен модуль програми повинен бути максимально незалежним. Модулі повинні спілкуватися між собою виключно за допомогою інтерфейсів і не повинні нічого знати про реалізацію один одного;

д) мінімізація змін коду при розробці. Це заощаджує не тільки час розробки, це економить також час тестування, так як немає необхідності повторно тестувати те, що вже було одного разу протестовано. Для виконання такого принципу необхідно дотримуватися правила, що для модифікації кожного класу має бути лише одна причина – класи повинні виконувати дуже конкретні місії;

е) абстракція. Оскільки програма – це модель реального життя, то класи повинні відображати абстрактну модель об'єкта або концепції реального життя, їх інтерфейс повинен бути інтуїтивно зрозумілий;

ж) поліморфізм. Поліморфізм має на увазі, що модуль повинен працювати по-різному в залежності від контексту його роботи. У той же час, користувач не повинен про це дбати, модуль повинен сам вибрати коректну поведінку;

з) успадкування. Незважаючи на те, що спадкування широко використовується в об'єктно-орієнтованому програмуванні, необхідно обережно його використовувати. Причиною тому є те, що поведінка класів під час виконання є комбінацією деякої кількості налаштувань. Для того, щоб забезпечити правильну роботу класів в такій ситуації, необхідно множинне успадкування, що є неприпустимим в багатьох сучасних мовах програмування. В даному випадку розумніше застосовувати шаблон програмування «Агрегування» замість наслідування. У більшості випадків множинне успадкування від класів може бути замінено множинним успадкування інтерфейсів і агрегації їх реалізації;

и) використання шаблонів проектування. Існує ряд однотипних завдань, з якими програмісти часто зустрічаються в процесі своєї роботи. Також широко застосовуються рішення, які називаються шаблонами проектування. Велика кількість цих шаблонів є строго вузькоспрямованими і має ряд своїх переваг та недоліків. Тому не варто використовувати перший ліпший шаблон, що підходить для конкретного випадку – слід проаналізувати всі такі шаблони.

Методи перетворення слабоструктурованого коду до бажаної архітектури

Як вже зазначалося, потреба в зміні архітектури може виникнути в різних ситуаціях. Для реалізації задачі зміни архітектури необхідно розробити методичний і керований підхід до її вирішення, а також супроводжуючих її інструментальних засобів.

Одним з найбільш успішних підходів до зміни існуючого програмного забезпечення є рефакторинг – підхід, що базується на систематичних трансформаціях вихідного коду. Рефакторинг – це такі зміни у внутрішній структурі ПЗ, які мають на меті полегшити розуміння його роботи та спростити модифікацію, не змінюючи існуючої поведінки. У звичному розумінні розробки ПЗ спочатку створюється дизайн системи, а потім пишеться її код. Згодом код модифікується, і цілісність системи, відповідність її структури до початкового дизайну поступово погіршується. Подальший розвиток системи поступово деградує від спрямованої, проектованої діяльності до «хакерства». Рефакторинг є протилежним підходом. З його допомогою можна взяти слабоструктурований, хаотичний проект і переробити його в добре спроектований код. Кожен крок цього процесу надзвичайно простий. Наприклад, кроком може стати переміщення поля або методу з одного класу до іншого, розщеплення класу і т.д. Однак сумарний ефект таких невеликих змін виявляє кумулятивну дію і може радикально поліпшити проект. Процес рефакторинга є прямою протилежністю поступової деградації коду системи [2].

При документації та каталогізації методів рефакторинга прийнято використовувати напівформальну нотацію, в якій кожен з методів описаний так званим шаблоном проектування. Будь-який шаблон проектування описує і іменує типову задачу, яка постійно виникає у роботі, а також принцип її вирішення, причому таким чином, що це рішення можна використовувати потім знову і знову. Шаблон іменує, абстрагує і ідентифікує ключові аспекти структури загального рішення [4]. Крім того, шаблони проектування формують словник рішень для цієї проблемної області і дозволяють двом фахівцям в цій області ідентифікувати типові рішення і розуміти один одного, не пояснюючи кожен раз суть самих рішень.

Рефакторинг об'єктно-орієнтованого коду зарекомендував себе як ефективний спосіб вирішення задач еволюції і супроводу програм. Проте в даний час практично не існує досліджень, присвячених рефакторингу на більш високому рівні абстракції – рівні архітектури ПЗ. Відповідно, викликає значний інтерес перенесення даної методології на більш високий рівень абстракції. Загальною метою зусиль по розробці та стандартизації методики зміни архітектури, а також інструментальних засобів є підтримка цієї методики та отримання керованого і передбачуваного процесу перетворення архітектури.

Є деякі відмінності архітектурного та класичного рефакторингів.

При перенесенні методики рефакторинга на рівень архітектури існує ряд особливостей, які обумовлює зміна зовнішнього вигляду методу.

Об'єкти. При переході від класичного рефакторинга до архітектурного змінюються об'єкти, з якими йде робота. У класичному рефакторингу сутностями, з якими йде робота, є такі елементи, як клас, екземпляр класу. Архітектурний рефакторинг застосовується до систем і компонентів. У різних типах рефакторинга розрізняються також і види зв'язків, що виникають між об'єктами.

Масштаби змін. Класичний рефакторинг застосовується у суттєво менших масштабах – зазвичай наслідки застосування окремого шаблону класичного рефакторингу обмежується кількома файлами. Шаблон архітектурного рефакторинга застосовуються до компонентів архітектури. При перенесенні цих шаблонів назад з рівня структурної моделі на програмний код зміни можуть торкнутися істотно більшого обсягу існуючого коду.

Опис змін. Методи класичного рефакторинга можна проілюструвати статичним моделями мови та фрагментами коду. Для опису архітектурного рефакторинга представляється більш зручним використовувати спеціалізовані структурні моделі та інструментальні засоби підтримки зворотної інженерії.

Тестування. Трансформації можна вважати коректними, якщо вони не призводять до змін поведінки програмної системи в цілому. Наявність досить повного набору модульних (unit) тестів дозволяє переконатися в коректності трансформацій. Саме це робить модульне тестування одним із ключових аспектів класичного рефакторинга. Для архітектурного рефакторинга також постає завдання автоматичної перевірки коректності трансформацій. Звичайно, наявність модульних тестів підвищує впевненість розробника в коректності проведених змін навіть при застосуванні архітектурного рефакторинга. Тим не менше, на даний момент невідомо, наскільки ефективно модульне тестування працює з методами рефакторинга архітектури. Це, зокрема, обумовлено різницею в масштабах змін при класичному та архітектурному рефакторингах.

Архітектурний рефакторинг поділяють на певні фази. Специфічною рисою рефакторинга архітектури є те, що для досягнення проміжних цілей, що виникають в ході архітектурного рефакторинга, як правило, доводиться виконувати більше одного кроку. Ці кроки відносяться до різних фаз рішення поставлених архітектурних завдань. Можна умовно виділити наступні фази архітектурного рефакторинга:

а) фаза «розкопки» архітектури;

б) фаза трансформації архітектури;

в) фаза семантичного аналізу підсистем;

г) фаза проеціювання змін моделі на програмний код.

Більш докладний розгляд фаз архітектурного рефакторинга доцільно почати саме з проеціювання змін моделі на програмний код.

Проекція змін моделі на програмний код. Як уже зазначалося, для опису архітектурного рефакторинга представляється доцільним використання структурних моделей. Моделі витягуються з програмного коду автоматично, і кожному елементу моделі відповідає деяка множина символів вихідного коду програмної системи. Таким чином, редагування моделі, обумовлене застосуванням кроків рефакторинга архітектури, може (і часто повинно) бути спроектованим на реальний програмний код системи. Дійсно, при проектуванні видалення блоків з моделі необхідно визначити перелік рядків і файлів, що відповідають віддаленому блоку в програмному коді. Після цього необхідно видалити з програмного проекту виявлені рядки та файли. При проектуванні на код перенесення блоку в моделі переносяться відповідні рядки і файли у вихідному коді програмної системи і т.д.

Проекція на програмний код системи кроків, виконаних у ході деякого архітектурного рефакторинга, хоча і є суто механічною дією, тим не менш, дозволяє отримати практичну вигоду з проведеного аналізу. Вироблені таким чином трансформації можна розглядати як архітектурно керований рефакторинг програмного коду.

«Розкопки» архітектури. Кроки, що відносяться до цієї фази, характеризуються тим, що відповідні дії, що застосовуються до моделі, не орієнтовані на подальшу проекцію на програмний код. Вони потрібні тільки для розуміння і структуризації моделі.

Трансформація архітектури. Для кроків, що відносяться до трансформації архітектури, на відміну від кроків фази «розкопки», типово подальша проекція їх на реальний код програмної системи. Кроки цієї фази чітко пов'язані з реальною модифікацією коду системи і, в кінцевому рахунку, орієнтовані на його поліпшення. Слід також зазначити, що частина методів рефакторинга архітектури не може бути чітко віднесена до однієї з названих категорій (розкопки та трансформація). На практиці це означає, що рішення про проектуванні цих кроків на код приймає розробник, керуючись поставленим завданням.

Семантичний аналіз підсистем. Як правило, між кроками описаних вище фаз архітектурного рефакторинга робляться кроки, які можна віднести до фази семантичного аналізу підсистем: по ходу трансформацій часто постає завдання виявлення смислового навантаження підсистем. Для вирішення подібних задач, навіть у першому наближенні, часто доводиться дослідити реальний програмний код (тут знову ж таки допомагає точність моделі), аналізувати сигнатури функцій та коментарі, а при відсутності останніх і сам код функцій. Завдання фахівців, залучених до процесу архітектурного рефакторинга – по можливості мінімізувати обсяг семантичного аналізу (наприклад, шляхом видалення допоміжних блоків) та зробити його більш послідовним та спрямованим.

Приклад застосування шаблону виділення прошарків

В роботі [5] описується випадок, коли знайдені не суворі прошарки дозволили виявити архітектур-

ний дефект у програмній системі toolbus, який суттєво ускладнював реалізацію завдання, поставленого перед розробниками. Програмна система toolbus представляє для своїх клієнтів механізми комунікації. У системі використовується набір стандартних повідомлень, які були реалізовані на основі сокетів операційної системи та спеціалізованого текстового протоколу для обміну даними сокетами (на рис. 1 – за це відповідає прошарок "Layer 2: Protocol Implementation"). Щоб ізолювати тонкощі текстового протоколу від розробників, був реалізований API системи, що приховує як її сокети, так і коди команд, які через них передаються, за набором процедур мови програмування ("Layer 3: Toolbus API"). Однак у процесі розвитку в систему були додані допоміжні механізми ("Layer 4: Applications"). Саме ці механізми й порушували строгість шаруватої структури, оскільки вони зверталися до рівня роботи сокетів в обхід визначеного API системи (зв'язок від "Layer 4" до "Layer 3").

Оскільки перед розробниками стояло завдання зміни текстового протоколу, який використовується в системі для обміну через сокети, подібний дефект погрожував переробкою всіх стандартних механізмів системи. Архітектурний рефакторинг дозволив уникнути настільки неприємної ситуації і згодом значно скоротив час, необхідний безпосередньо на кодування.

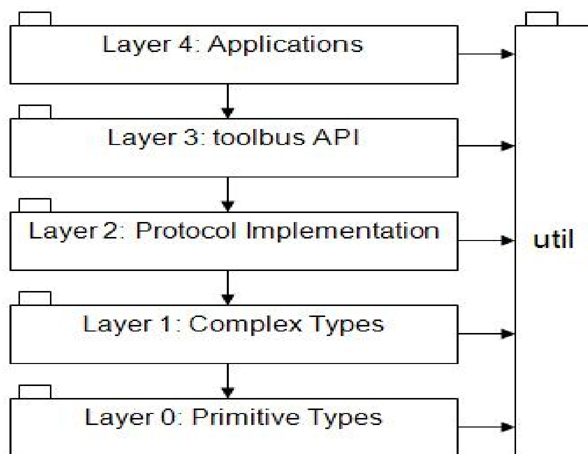

Рис. 1. Архітектура toolbus

Аналіз архітектури і поглинаючі прошарки. Характерним прикладом, що демонструє користь пом'якшення умов для пошуку прошарків з метою аналізу архітектури є ситуація, що виникла при аналізі архітектури Apache James 2.2. Apache James – це реалізований на Java потужний поштовий сервер масштабу підприємства, який підтримує всі найбільш поширені сучасні поштові протоколи і надає ряд додаткових можливостей. Після того, як для сервера Apache James була побудована структурна модель, і в ній були виділені основні підсистеми, була зроблена спроба виділити в моделі не суворі прошарки.

При виділенні прошарків було дозволено поглинання сильно зв’язаних компонентів. У результаті була отримана схема, представлена на рис. 2. Ця схема цілком обдумана: на прошарку 1 виявилися блок Constants.java, що містить константи, що використовуються у всій системі, і блок Core, який містить основні типи даних системи, такі як Message – повідомлення, MessageHeader – заголовок повідомлення та інші. Таким чином, прошарок 1 містить основні визначення системи. Прошарок 2 містить authorization – систему контролю доступу, mailrepository – репозиторій листів, mailstore адаптер для mailrepository створений для уніфікації роботи з різними системами зберігання та управління повідомленнями, в тому числі він може служити для приховування і mailrepository. Прошарок 3 – фактично, основний, оскільки містить власне ядро поштового сервера. На прошарку 4 розташовуються допоміжні механізми роботи з поштовим сервером – системи віддаленого адміністрування remotemanager і fetch – яка полегшує отримання листів з сервера.

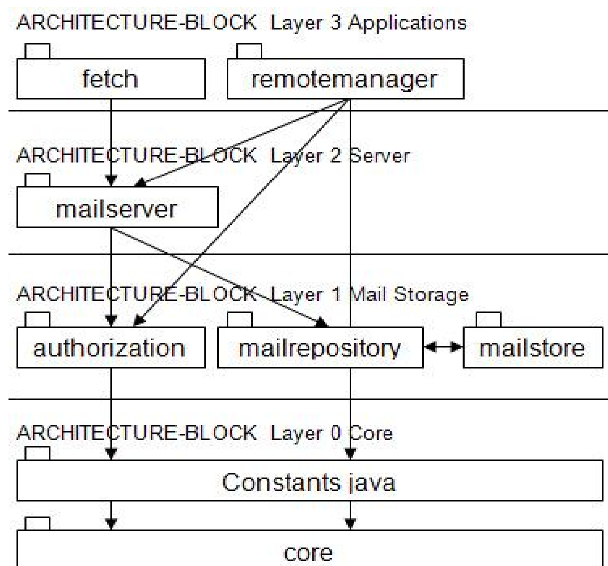

Рис. 2. Схема архітектури Apache James
з поглинанням сильно зв’язаних компонентів

З іншого боку, спроба виділення не суворих прошарків, але без поглинання сильно зв’язаних компонентів дала набагато більш слабкий результат. Виділені прошарки представлені на рис. 3.

Через те, що в системі існує сильно зв’язаний компонент, до якого входять блоки mailrepository і mailstore, на «горище» потрапило більшість прошарків системи, а дати прошаркам яке-небудь виразне визначення виявилося неможливо.

Необхідно відзначити, що сильно зв’язаний компонент, що містить mailrepository і mailstore, свідчить про деяку архітектурну проблему. Як вже говорилося, mailstore – виступає як адаптер для mailrepository. Зв'язок що йде від mailrepository до адаптера свідчить про те, що зміна адаптера призве-

де до зміни адаптованості об'єкта, що знижує гнучкість архітектури програмної системи.

Усі перетворення коду виконуються за допомогою рефакторингу. Починається цей процес з «розкопки» архітектури, коли методи рефакторингу використовуються для надання коду зрозумілого вигляду. Це необхідно для того, щоб можна було оцінити поточне становище системи. Далі наступає етап трансформації архітектури, який призводить до змін коду програми. Періодично виконується семантичний аналіз системи, який дозволяє контролювати роботи, які ведуться з кодом та своєчасно реагувати на помилкові дії.

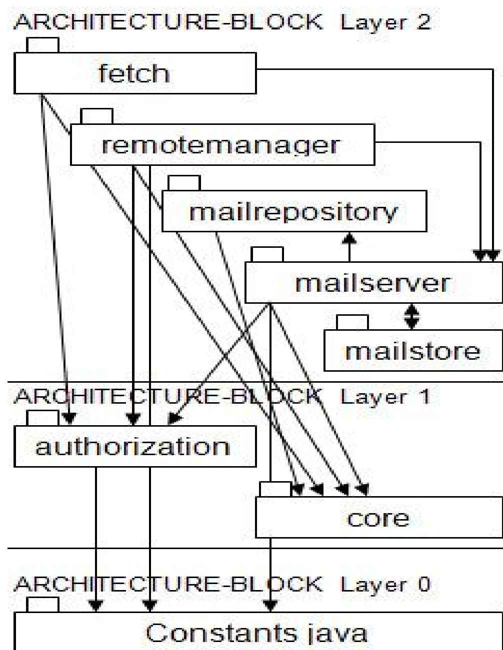

Рис. 3. Схема архітектури Apache James
без поглинанням сильно зв'язаних компонентів

Усі роботи з кодом виконуються за допомогою найпростіших рефакторингів. Такі рефакторинги відомі майже будь-якому сучасному програмісту та дозволяють досить просто, невеликими кроками виконувати значні перетворення коду [2].

Висновки

В результаті проведених досліджень були розглянуті прояви слабоструктурованого коду та запропонований підхід, дотримуючись якого можна розробити структуровану архітектуру, яка відповідає необхідним критеріям.

Користуючись наведеними ознаками слабоструктурованого коду можна зробити оцінку стану програмної системи та прийняти рішення про її модифікацію з метою покращення.

Далі, користуючись загальноприйнятими та випробуваними підходами до організації архітектури системи, можна виробити нову, більш вдалу структуру програми.

Для перетворення існуючого коду до бажаного архітектурного стану слід використовувати рефакторінг, як ефективний спосіб контрольованої модифікації структури програмної системи.

В результаті був розроблений метод перетворення слабоструктурованого коду до бажаного архітектурного стану з використанням техніки рефакторингу.

Список літератури

1. *Deursen van A. Research Issues in The Renovation of Legacy Systems / A. van Deursen, P. Klint, C. Verhoef. – CWI research report P9902, April 1999.*

2. *Фаулер М. Рефакторинг: Улучшение существующего кода / М. Фаулер. – Спб.: Символ, 2003. – 320 с.*

3. *Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – Спб.:Питер, 2001. – 544 с.*

4. *Alexander C. A Pattern Language / C. Alexander, S. Ishikawa, M. Silverstain, M. Jakobson, I. Fiksdahl-King, S. Angel. – New York: Oxford University Press, 1977.*

5. *Ксензов М.В. Рефакторинг архитектуры программного обеспечения / М.В. Ксензов. – М.: ИСП РАН, 2004. – Препринт 4. – 88 с.*

Надійшла до редколегії 25.05.2009

Рецензент: д-р техн. наук, проф. В.М. Тупкало, Центральний НДІ навігації і управління, Київ.

МЕТОД ПРЕОБРАЗОВАНИЯ АРХИТЕКТУРЫ БИЗНЕС-ЛОГИКИ В СЛАБОСТРУКТУРИРОВАНЫХ ПРОГРАММНЫХ СИСТЕМАХ

А.Л. Ерохин, М.О. Мончак

В статье рассмотрены признаки структурированной и неструктурированной архитектуры программных систем и метод преобразования структуры системы с помощью архитектурного рефакторинга. Данный метод позволяет производить эффективные изменения кода с целью приведения его к желаемому архитектурному состоянию.

Ключевые слова: *архитектура системы, структура программы, архитектурный рефакторинг.*

THE METHOD OF TRANSFORMING THE ARCHITECTURE OF BUSINESS LOGIC IN WEAK-STRUCTURED SOFTWARE SYSTEMS

A.L. Yerokhin, M.O. Monchak

The article deals with features of structured and unstructured architecture of software systems and the method of transforming the structure of the system with the help of the architectural refactoring. This method allows for efficient code changes to bring it to the desired architectural state.

Keywords: *architecture of the system, program structure, architectural refactoring.*